CompSci 230
Software Construction

Java Implementation, Part 2

S1 2015

Agenda

- Topics:
  - Packages: why and how?
  - Visibility, and its effect on inheritance
  - Static and dynamic typing
  - Object conversion, casting
- Reading:
  - In The Java Tutorials:
    - Controlling Access to Members of a Class, in the Classes and Objects Lesson
    - The Packages Lesson
    - Inheritance, in the Interfaces and Inheritance Lesson
- Reference:
  - Conversions and Contexts, in the Java Language Specification, Java SE 8 Edition, 2015-02-13.

Packages

- **Definition:** "A *package* is a namespace that organizes a set of related classes and interfaces."
- **Explanation:** "Conceptually you can think of packages as being similar to different folders on your computer.
  - You might keep HTML pages in one folder, images in another, and scripts or applications in yet another.
  - Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense
    - to keep things organized by placing related classes and interfaces into packages."

http://docs.oracle.com/javase/tutorial/java/concepts/package.html

Packages (alternate definition)

- **Rationale:** "To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages."
- "**Definition:** A *package* is a grouping of related types providing access protection and name space management."
  - Note that *types* refers to classes, interfaces, enumerations, and annotation types.
  - Enumerations and annotation types are special kinds of classes and interfaces, respectively, so
    - *types* are often referred to in this lesson simply as *classes and interfaces*."

http://docs.oracle.com/javase/tutorial/java/package/packages.html

Creating a Package

- "To create a package, you
  - choose a name for the package (naming conventions are discussed in the next section) and
  - put a package statement with that name at the top of *every source file* that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.
- "The package statement (for example, **`package graphics;`**) must be the first line in the source file.
  - There can be only one package statement in each source file, and it applies to all types in the file."

http://docs.oracle.com/javase/tutorial/java/package/createpkgs.html

One public type per file!

- "If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file.
  - For example, you can
    - define **`public class Circle`** in the file **`Circle.java`**,
    - define **`public interface Draggable`** in the file **`Draggable.java`**,
    - define **`public enum Day`** in the file **`Day.java`**, and so forth.
- "You can include non-public types in the same file as a public type
  - (this is strongly discouraged, unless the non-public types are small and closely related to the public type),
  - but only the public type will be accessible from outside of the package.
  - All the top-level, non-public types will be *package private*."
- This rule makes it easy for the class loader, and the human programmer, to find the definition for a public type.
  - The name of a package determines the directory in which the files of this package *should* be stored.
  - The name of a public type determines the name of the file in which the type's definition *must* be found."

http://docs.oracle.com/javase/tutorial/java/package/createpkgs.html

The default package

- "If you do not use a package statement, your type ends up in an unnamed package.
  - Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process.
  - Otherwise, classes and interfaces belong in named packages."

http://docs.oracle.com/javase/tutorial/java/package/createpkgs.html

Package naming conflicts

- "With programmers worldwide writing classes and interfaces using the Java programming language,
  - it is likely that many programmers will use the same name for different types.
  - In fact, the previous example does just that: It defines a **`Rectangle`** class when there is already a **`Rectangle`** class in the **`java.awt`** package.
  - Still, the compiler allows both classes to have the same name if they are in different packages.
- The fully qualified name of each **`Rectangle`** class includes the package name.
  - That is, the fully qualified name of the **`Rectangle`** class in the **`graphics`** package is **`graphics.Rectangle`**, and
  - the fully qualified name of the **`Rectangle`** class in the **`java.awt`** package is **`java.awt.Rect`**angle.
- This [syntax for fully qualified names] works well unless two independent programmers use the same name for their packages.
  - What prevents this problem [of name conflict]? Convention."

http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

Package naming conventions

- "Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
  - Companies use their reversed Internet domain name to begin their package names
    - for example, **`com.example.mypackage`** for a package named **`mypackage`** created by a programmer at example.com.
  - Name collisions that occur within a single company need to be handled by convention within that company,
- Packages in the Java language itself begin with **`java.`** or **`javax.`**"

http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

External references

- “To use a public package member from outside its package, you must do one of the following:
  - Refer to the member by its fully qualified name
  - Import the package member
  - Import the member’s entire package.
- The fully qualified name for class `C` in package `p` is `p.C`
  - To import class `C` from package `p`, you write `import p.C`
    - [This allows you to refer to the class as `C` rather than `p.C`]
  - To import an entire package `p`, you write `import p.*`
  - Each is appropriate for different situations…”
- If you import a package which defines a class C then your code may refer to it by its simple name, rather than its fully-qualified name, unless this name is ambiguous:
  - “If a member in one package shares its name with a member in another package and both packages are imported, you must refer to each member by its qualified name. ”

http://docs.oracle.com/javase/tutorial/java/package/usepkgs.html

Warning: Packages are not Nested!

- "At first, packages appear to be hierarchical, but they are not.
  - For example, the Java API includes a `java.awt` package, a `java.awt.color` package, a `java.awt.font` package, and many others that begin with `java.awt`.
  - However, the `java.awt.color` package, the `java.awt.font` package, and other `java.awt.xxxx` packages are not included in the `java.awt` package.
  - The prefix `java.awt` (the Java Abstract Window Toolkit) is used for a number of related packages to make the relationship evident, but not to show inclusion."

http://docs.oracle.com/javase/tutorial/java/package/usepkgs.html

Control of the "Name Space"

- Java gives you two major ways to control the "name space" of your programs:
  - You control the import of external names (by your `import` statements)
  - You control the export of your names (by restricting visibility, in packages and in inheritances).

Visibility Rules

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- “The first data column indicates whether the class itself has access to the member defined by the access level.
- The second column indicates whether [other] classes in the same package as the class (regardless of their parentage) have access to the member.
- The third column indicates whether subclasses of the class declared outside this package have access to the member.
- The fourth column indicates whether all classes have access to the member.”

[The Java Tutorials, Controlling Access to a Member or Class]

Tips on Choosing an Access Level

- "If other programmers use your class, you want to ensure that errors from misuse cannot happen.
  - Access levels can help you do this.
- "Use the most restrictive access level that makes sense for a particular member.
- "Use private unless you have a good reason not to.
- "Avoid public fields except for constants.
  - (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.)
  - Public fields tend to link you to a particular implementation and limit your flexibility in changing your code."

[The Java Tutorials, Controlling Access to a Member or Class]

Inheritance and Visibility

- Every subclass will
  - inherit all superclass members that are declared as **`public`** or **`protected`** .
- By contrast,
  - **`private`** members are not inherited (but may be accessible through **`super`**.)
  - The default visibility is "package-private" – inherited by subclasses within the same package, but not inherited by subclasses that are declared outside the package.
- No subclass can
  - override **`static`** methods, or
  - override **`final`** methods.
- Any subclass may
  - add new members (= fields or methods), or
  - override any non-**`static`**, non-**`final`** method in the superclass.
- Recall from the previous slides: We say a method is overridden in a subclass, if any of its superclasses has a method of the same signature (= name, plus the number and types of parameters) and return type.
  - Note that overriding does not absolutely prevent access. A reference to the superclass member is still possible (e.g. with **`super`**) if this member is visible.

Statically or Dynamically typed

- Programming languages generally offer some sort of type system, and can be described as being either statically typed or dynamically typed
- With a statically typed language, compile-time checks are carried out to determine whether variable usage is valid. In Java:

```
int x = 10;        ✓

x = "Hello";       ✗
```

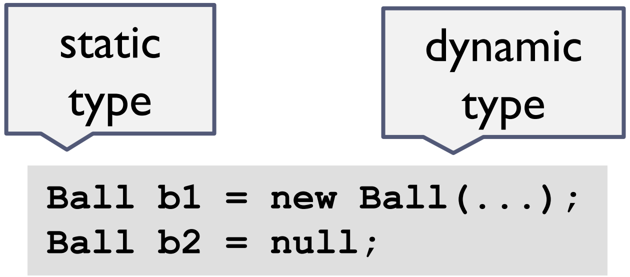
- In a dynamically typed language, variables are not associated with a type and are simply names that can be assigned arbitrary values. In Python:

```
x = 10             ✓

x = "Hello"        ✓
```

Java - a statically typed language

- Every variable name is bound
  - to a static type (at compile time, by means of a data declaration), and
  - either to a dynamic type or `null`, depending on its current value



static type

dynamic type

```
Ball b1 = new Ball(...);
Ball b2 = null;
```

- The type restricts the values that can be bound to this variable.
  - `int x = 2.3;`
- The type also restricts the messages that can be sent using the variable.
  - `int x = 2;   (Vector) x.add(0x37);`
- Restrictions are checked at compile-time.
  - The compiler will not issue code if it detects a violation.
  - Java is a "type-safe" language: its compile-time checking restricts the amount of damage that can be done by careless or malicious programmers.

Static Typing Restrictions

- A reference variable of static type T can refer to an instance of class T or to an instance of any of T's subclasses.
  - A type is a restriction on the values that can be taken by a variable, and a subclass is a stricter restriction – so there can be no type error when a value in a subtype of T is assigned to a variable of type T.
- Through a reference variable of static type T, the set of messages that can be sent using that variable are the methods defined by class T and its superclasses.
  - This typing rule allows inherited methods to be accessed via T, in contexts where the names of these methods are visible.
  - There might be many subclasses of T, each defining different methods with the same name – so T can't be used to refer to any of these subclass methods.
- Recall: a variable's static type is fixed at compile time,
  - but its dynamic type may vary at run-time.
- To learn more about static & dynamic typing from a Java perspective, see Java Virtual Machine Support for Non-Java Languages

Example: Static Binding of Instance Variables

```
class Base {
  public int x = 10;
}
```

```
public class Derived extends Base {
  public int y = 20;
}
```

```
//Case 1:
Base b1 = new Base();
System.out.println("b1.x=" + b1.x);
```

Instance variable x in Base.

```
b1.x=10
```

```
//Case 2:
Derived b2 = new Derived();
System.out.println("b2.x=" + b2.x);
System.out.println("b2.y=" + b2.y);
```

b2 has static type Derived, and dynamic type Derived.

Instance variable x in Derived: inherited from Base

```
b1.x=10
b2.y=20
```

```
//Case 3:
Base b3 = new Derived();
System.out.println("b3.x=" + b3.x);
// System.out.println("b3.y=" + b3.y);
```

b3 has static type Base, and dynamic type Derived.

```
b3.x=10
```

There is no y declared in the Base class – this won’t compile!

Static Binding – Hiding a Field

- “Within a class, a field that has the same name as a field in the superclass hides the superclass's field,
  - even if their types are different.
- “Within the subclass, the field in the superclass cannot be referenced by its simple name.
  - “Instead, the field must be accessed through **`super`**, which is covered in the next section.
- “Generally speaking, we don't recommend hiding fields as it makes code difficult to read.” [The Java Tutorials]

```
class Base {
  public int x = 10;
}
```

```
public class Derived extends Base {
  public String x = “20”;
}
```

```
Base b3 = new Derived();
System.out.println("b3.x=" + b3.x);
```

Review: Fields & Variables

- The Java Tutorials makes a careful distinction between fields and variables.
  - Not many programmers use these terms carefully.
  - You won't understand the Java Tutorials, in full technical detail, unless you understand its definitions!
- In the Variables page of the Language Basics Lesson:
  - **"Instance Variables (Non-Static Fields)** Technically speaking, objects store their individual states in 'non-static fields', … also known as *instance variables* …
  - **"Class Variables (Static Fields)** A *class variable* is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.
  - **"Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. … There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
  - **"Parameters** … The important thing to remember is that parameters are always classified as 'variables' not 'fields'. … [In addition to methods,] other parameter-accepting constructs … [include] constructors and exception handlers …"

Dynamic Binding

- If a method is overridden, then the compiler may not be able to resolve a reference to that method.
- The runtime search for an overridden method begins with the dynamic type.
  - If this type doesn't implement the method (i.e. it neither introduces nor overrides the method), then the search progresses up the hierarchy, until the method is found.
  - Static type-checking ensures that an implementation will be found (unless the class was changed, and re-compiled, after the type-check.)

```
class Base {
  public void f() { ... }
  public void g() { ... }
}
```

```
Derived b2 = new Derived();
Base b3 = new Derived();
```

Dynamic type

```
public class Derived extends Base {
  public void g() { ... }
  public void h() { ... }
}
```

```
b2.f();
```

Inherited: invoke f() in Base

```
b2.g();
```

Overridden: invoke g() in Derived

```
b2.h();
```

Introduced: invoke h() in Derived

Dynamic Binding

- If a method is overridden, then the compiler may not be able to resolve a reference to that method.
- The runtime search for an overridden method begins with the dynamic type.
  - If this type doesn't implement the method (i.e. it neither introduces nor overrides the method), then the search progresses up the hierarchy, until the method is found.
  - Static type-checking will ensure that an implementation will be found -- unless the class was changed, and re-compiled, after the type-check!

```
class Base {
  public void f() { ... }
  public void g() { ... }
}
```

```
public class Derived extends Base {
  public void g() { ... }
  public void h() { ... }
}
```

```
Derived b2 = new Derived();
Base b3 = new Derived();
```

Dynamic type

```
b3.f();
```

Inherited: invoke f() in Base

```
b3.g();
```

Overridden: invoke g() in Derived

```
b3.h();
```

Out of scope: compile-time error

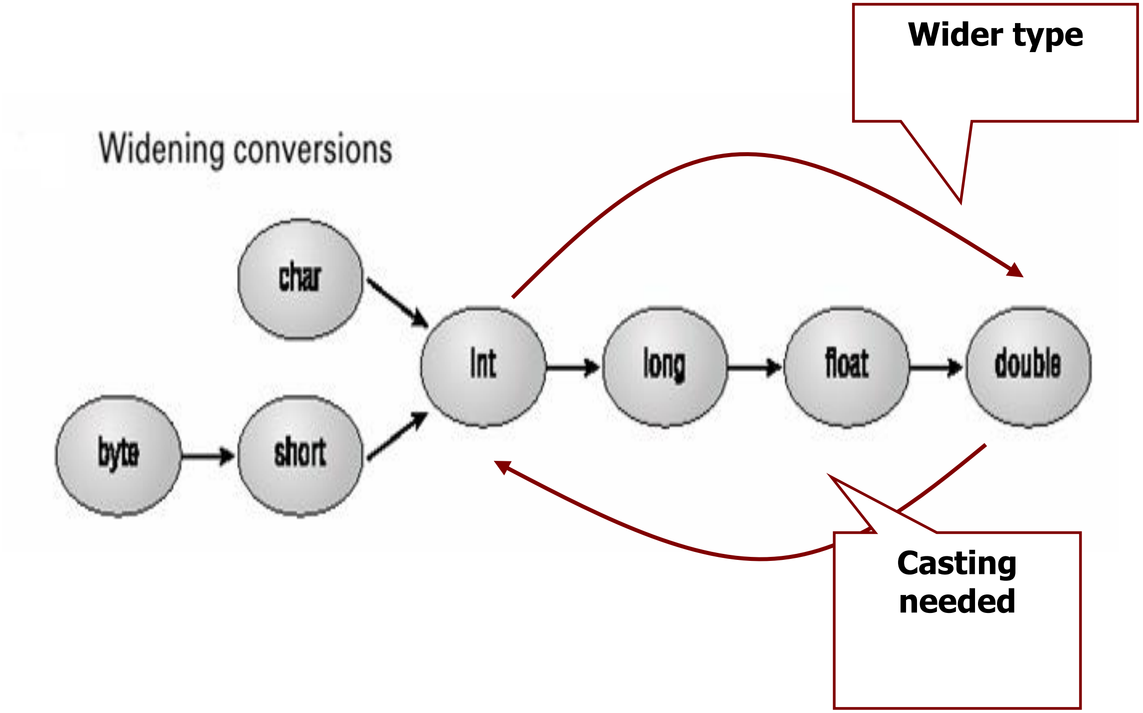
Conversions of Primitive Types

- Widening conversions
  - Wider assignment, e.g. `int i = 2; float x = i;`
  - Wider casting, e.g. `int i = 2; double d = (double) i;`
    - Explicitly casting can make your code more readable
- Narrowing conversions
  - Narrow assignment
    - a compile-time error!

```
float f = 2.0;
int i = f;
```

  - Narrow casting
    - a loss of information!

```
float f = 2.0;
int i = (int) f;
```

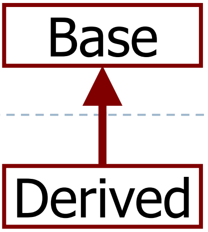


Wider type

Casting needed

Object Type Conversions

Wider type



- Widening conversions
  - Wider object reference assignment conversion (allowed)
  - Wider object reference casting (optional: improves readability)

```
Base b = new Base();
Derived d = new Derived();
Base b1, b2;
System.out.println(d.y);
```

```
b1 = d;
//System.out.println(b1.y);
```

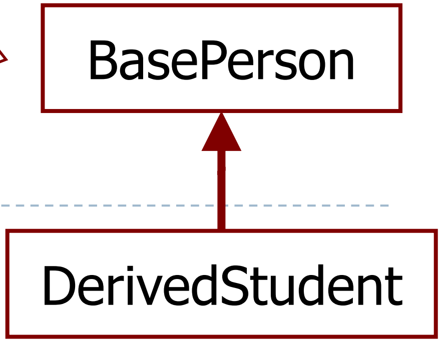
Assignment conversion - OK
But no access to fields in Derived!

```
b2 = (Base) d;
//System.out.println(b2.y);
```

Widening with explicit cast - Better
Still no access to fields in Derived!

Object Types

Wider type



- Narrowing conversions
  - Narrow object reference assignment – Compile-time error!
  - Narrow object reference casting – no compilatation error, but…
    - The cast may throw an error at run-time, to avoid assigning an out-of-range value!

```
Base b = new Base();
Derived d = new Derived();

Derived d1, d2, d3;
```

```
d1 = b;
```

A compile-time error

```
d2 = (Derived) b;
```

Compile-time OK, Run-time ERROR
b is an instance of class Base, not Derived!

```
java.lang.ClassCastException: Base
```

```
Base d_as_b = new Derived();
d3 = (Derived) d_as_b;
```

Compile-time OK: Derived is a narrower (more refined) type

Run-time OK:
d_as_b is an instance of Derived

Overriding, hiding, and overloading methods

- "An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method."
- "If a subclass defines a class method with the same signature as a class method in the superclass, the method in the subclass *hides* the one in the superclass.
  - "The distinction between hiding and overriding has important implications.
    - The version of the overridden method that gets invoked is the one in the subclass.
    - The version of the hidden method that gets invoked depends on whether it is invoked from the superclass or the subclass."
- "Overloaded methods are differentiated by the number and the type of the arguments passed into the method."
  - "The compiler does not consider return type when differentiating methods, so you cannot declare two methods [in the same class] with the same signature even if they have a different return type.
  - "**Note:** Overloaded methods should be used sparingly, as they can make code much less readable."

Review

- Topics:
  - Packages:
    - Why and how?
    - What conventions should you follow?
  - Four visibility keywords:
    - How do they affect the scope of access to a field or method?
  - Static and dynamic typing:
    - When do they occur?
    - What is "type-safety"?
  - Object conversion, casting:
    - What is allowed at compile-time?
    - What might happen at run-time?
    - How do they affect readability?